Assembly Language

Assembly Language: Human Readable Machine Language

Computers like ones and zeroes…

```
0001110010000110
```

Humans like symbols…

```
ADD    R0,R17 ; increment index reg.
```

An assembler is a program that translates symbols for instructions into actual machine instructions.

- ISA-specific
- Close relationship between symbols and insn-set
- Mnemonics for opcodes
- Labels for memory locations
- Additional operations (directives) for various tasks like allocating and initializing storage

Example Assembly Language Program

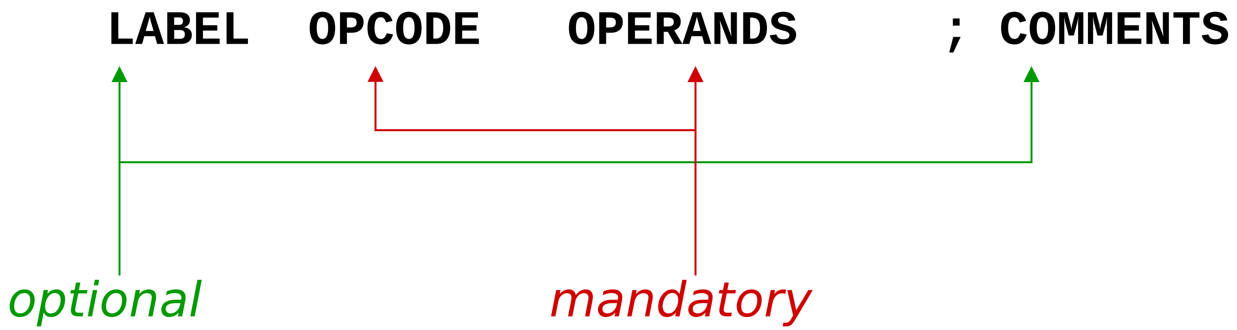
```
;This program multiplies the value in r17 (23) by the value in r16 (6)
; and places the result in r0. It will work on the atmega328P.
jmp entry

.org 0x100
entry:
        ldi r16, 6
        ldi r17, 23
        ldi r18, -1
        eor r0, r0
loop:
        add r0, r17
        add r16, r18
        brbc 1, loop
sink:
        rjmp sink
```

AVR Assembly Language Syntax

- Each line of a program is one of the following:
  - an instruction
  - an assember directive (or pseudo-op)
  - a comment
- Whitespace (between symbols) and case are ignored.
- Comments (beginning with “;”) are also ignored.

- An instruction has the following format:



```
LABEL  OPCODE   OPERANDS     ; COMMENTS
```

optional

mandatory

Opcodes and Operands

- Opcodes
  - reserved symbols that correspond to AVR instructions
    - ex: `add`, `eor`, `ldi`, `brbc`, …

- Operands
  - registers -- specified by Rn, where n is the register number
  - numbers – Hexadecimal indicated by 0x or $
  - label -- symbolic name of memory location
  - Operands separated by commas
  - number, order, and type correspond to instruction format
    - ex:

      ```
      add  r1,r3
      com r1
      ldi r31,0xff
      brbc 1,loop
      ```

Labels and Comments

- Label
  - placed at the beginning of the line
  - assigns a symbolic name to the address corresponding to line
    - ex:

```
loop: add r1,r3
      brvc loop
```

- Comment
  - anything after a semicolon is a comment
  - ignored by assembler
  - used by humans to document/understand programs
  - tips for useful comments:
    - Do what you feel is useful

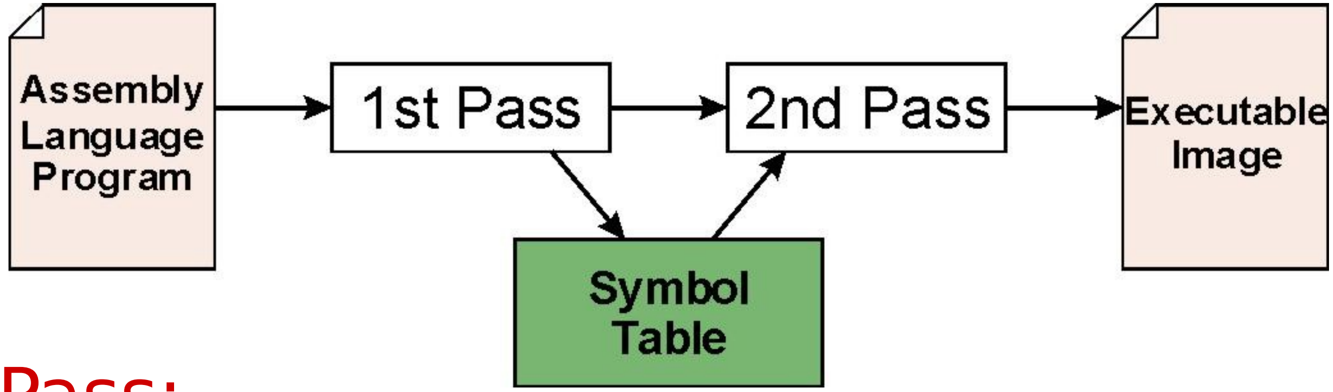
Assembler Directives

- Pseudo-operations
  - do not refer to operations executed by program
  - used by the assembler
  - look like instructions, but the “opcode” starts with dot
  - assembler-specific

Opcode	*Operand*	*Meaning*
.ORG	**address**	**starting addr of next insn in PMEM**
.BYTE	**expressions**	**Place bytes from exprs in code**
.SET	**symbol,expr**	**Set value of symbol to expression**
.FILL	**Repeat,size, value**	**allocate one word, initialize with value n**
.SECTION	**sectionname**	**Place following code into section sectionname**

Assembly Process

- Convert assembly language file (.asm) into an executable file (.hex) for the AVR.



- First Pass:
  - scan program file
  - find all labels and calculate the corresponding addresses;
    this is called the *symbol table*
- Second Pass:
  - convert instructions to machine language, using information from symbol table

First Pass: Constructing the Symbol Table

1. Initialize the location counter (LC) which keeps track of the address of the current instruction.
   - On AVR, LC is initialized to 0.

2. For each non-empty line in the program:
   a) If line contains a label, add label and LC to symbol table.
   b) Increment LC.
      - NOTE: If statement is `.BYTE` or `.FILL`, increment LC by the number of words allocated.

3. Stop when tend of file is reached.

- NOTE: A line that contains only a comment is considered an empty line.

Second Pass: Generating Machine Language

- For each executable assembly language statement, generate the corresponding machine language instruction.
  - If operand is a label, look up the address from the symbol table.

- Potential problems:
  - Improper number or type of arguments
    - ex:
      ```
      rcallr3
      ldi r0,0xff
      add r3,r3,128
      ```
  - Immediate argument too large
    - ex: `ori r1,0xdeadbeef`

Linking and Loading

- *Loading* is the process of copying an executable image into memory.
  - more sophisticated loaders are able to *relocate* images to fit into available memory
  - must readjust branch targets, load/store addresses

- *Linking* is the process of resolving symbols between independent object files.
  - suppose we define a symbol in one module, and want to use it in another
  - some notation, such as `.extern`, is used to tell the assembler that a symbol is defined in another module
  - linker will search symbol tables of other modules to resolve symbols and complete code generation before loading

Building An Assembly Language Program Using GNU Toolchain

- *avr-as –mmcu=atmega328p myfile.asm*
  - produces a.out
- *avr-ld –m avr5 –o myfile.elf a.out*
  - produces .elf file from a.out
- *avr-objcopy –O ihex –R .eeprom myfile.elf myfile.hex*
  - produces Intel .hex (ROM image) from .elf
- *ldino –P myfile.hex*
  - Programs the atmega328p on Arduino with contents of myfile.hex